Università di Bologna

Dipartimento di Informatica - Scienze e Ingegneria

Stereo Robot Perception

Computer Vision and Image Processing M

Lorenzo Severini

Contents

List of Figures

Chapter 1

Project Details

1.1 Objective

Given a video sequence taken by a stereo camera mounted on a moving vehicle, project's objective is to sense information concerning the space in front of the vehicle which may be deployed by the vehicle navigation system to automatically avoid obstacles.

1.2 Dataset

The input data consist of a pair of synchronized videos taken by a stereo camera (robotL.avi, robotR.avi), with one video concerning the left view (robotL.avi), the other the right view (robotR.avi). Moreover, the parameters required to estimate distances from stereo images are provided below:

- focale $f = 567.2$ pixel
- baseline $b = 92.226$ mm

In case of problems associated with reading the video files, students may try to install an Indeo CODEC, e.g. after downloading it from the following URL:
`http://downloadcenter.intel.com/Detail_Desc.aspx?strState=LIVE&ProductID=355&DwnldID=2846`

1.3 Functional Specifications

Sensing of 3D information related to the obstacles in front of the vehicle should rely on the stereo vision principle. Purposely, students should develop an area-based stereo matching algorithm capable of producing a dense disparity map for each pair of synchronized frames and based on the SAD (Sum of Absolute Differences) dissimilarity measure. For each pair of candidate corresponding points, the basic stereo matching algorithm consists in comparing the intensities belonging to two squared windows centred at the points. Such a comparison involves computation of either a dissimilarity (e.g. SAD, SSD) or similarity (e.g. NCC, ZNCC) measure between the two windows. As the matching process is carried out on rectified images, once a reference image is chosen (e.g. the left view), the candidates associated with

a given point need to be sought for along the same row in the other image (right view) only and, usually, within a certain disparity range which depends on the depth range one wishes to sense. Accordingly, given a point in the reference image, the corresponding one in the other image is selected as the candidate minimizing (maximizing) the chosen dissimilarity (similarity) measure between the windows. As such, the parameters of the basic stereo matching algorithm consist in the size of the window and the disparity range. In this project, students should choose the former properly, while the latter is fixed to the interval [0, 128].

The main task of the project requires the following steps:

- Computing the disparity map in a central area of the reference frame (e.g. a squared area of size 60x60, 80x80 o 100x100 pixels), so to sense distances in the portion of the environment which would be travelled by the vehicle should it keep a straight trajectory.

- Estimate a main disparity (d_{main}) for the frontal (wrt the camera) portion of the environment based on the disparity map of the central area of the reference frame computed in the previous step, e.g. by choosing the average disparity or the most frequent disparity within the map.

- Determine the distance (z, in mm) of the obstacle wrt to the moving vehicle based on the main disparities (in pixel) estimated from each pair of frames:

$$z(mm) = \frac{b(mm) \cdot f(pixel)}{d_{main}(pixel)}$$

- Generate a suitable output to convey to the user, in each pair of frame, the information related to the distance (converted in meters) from the camera to the obstacle. Moreover, an alarm should be generated whenever the distance turns out below 0.8 meters.

- Compute the real dimensions in mm (W, H) of the chessboard pattern present in the scene. Purposely, the OpenCV functions cvFindChessboardCorners and cvDrawChessboardCorners may be deployed to, respectively, find and display the pixel coordinates of the internal corners of the chessboard. Then, assuming the chessboard pattern to be parallel to the image plane of the stereo sensor, the real dimensions of the pattern can be obtained from their pixel dimensions (w, h) by the following formulas:

$$W(mm) = \frac{z(mm) \cdot w(pixel)}{f(pixel)}$$

$$H(mm) = \frac{z(mm) \cdot h(pixel)}{f(pixel)}$$

Moreover, students should compare the estimated real dimensions to the known ones (125 mm x 178 mm) during the first approach manoeuvre of the vehicle to the pattern, so to verify that accuracy becomes higher as the vehicle gets closer to the pattern. Students should also comment on why accuracy turns out worse during the second approach manoeuvre.

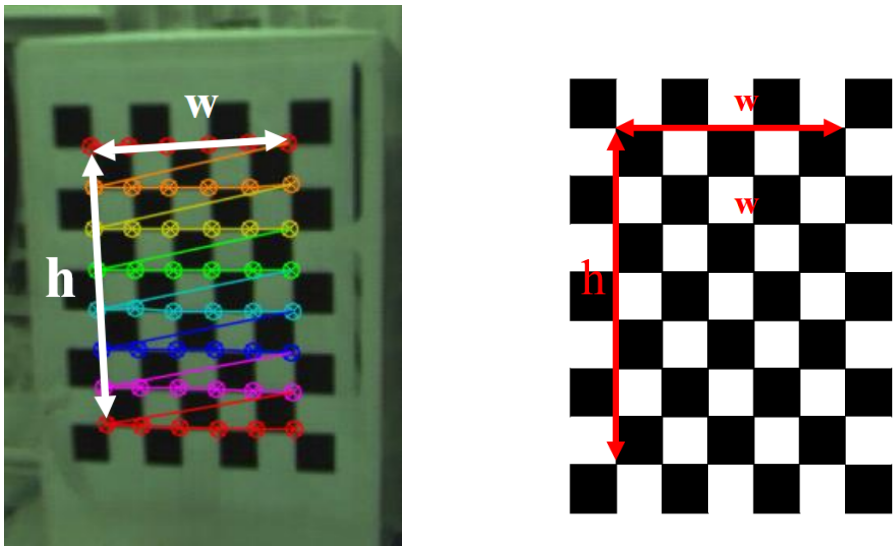

Figure 1.1: Respectively a chessboard capture from robot navifation and a 2D representation.

1.4 Possible improvements

Optionally, students may modify their software system to implement one (or more) of the following improvements.

- Modify the matching algorithm so to deploy a smaller disparity range, such as e.g. $[0+o, 64+o]$, with o being a suitable horizontal offset. This offset can be computed as that value allowing the main disparity, d_{main}, computed at the previous time instant to lie at the centre of the disparity range. Accordingly, as the vehicle gets closer to the obstacle, the horizontal offset increases, thus avoiding the main disparity to exceed the disparity range (and conversely, when the vehicle goes away from the obstacle).

- Instead of just a single main disparity, compute a set of disparities associated with the different areas of the obstacle, so to then estimate the distance from the camera for each part of the obstacle. For example, the image may be divided into a few large vertical stripes (5 in the exemplar left picture below), assuming the vertical lines parallel to the obstacle to be parallel to the image plane of the stereo sensor, and then estimate for each stripe a main disparity value (as done previously). Accordingly, one may create a planar view of the obstacle computing the angle between the horizontal lines parallel to the obstacle and the image plane of the stereo sensor (see the exemplar right picture below).

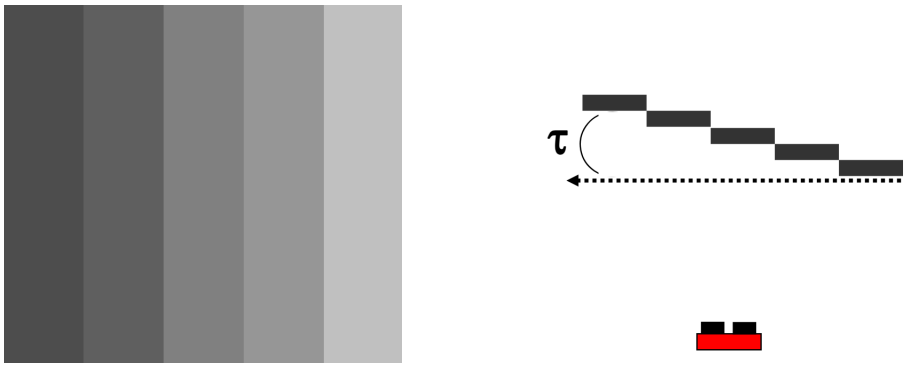

Figure 1.2: Stripes window and planar view representation

- Develop a more robust approach to estimate the main disparity (disparities) in order to filter out outliers. For example, ambiguous disparity measurements may be detected and removed by analysing the function representing the dissimilarity (similarity) measure along the disparity range or by computing disparities only at sufficiently textured image points (e.g, as highlighted by the well-known Moravec interest operator).

Chapter 2

Code

In this chapter we will review the code and motivate the choices made during the development. All of the requests in chapter 1 are accomplished in this project.

2.1 Libraries

The main libraries used in this project are `opencv-python` and `numpy`. Some minor libraries are imported for a better development, visualization and usage of the program, such as: `time`, `argparse`, `sys`.

There are also self-made "libraries": files where i stored code to make the main one more "readable":

- `utils.disparity_func` to separate the implemented disparity functions;
- `utils.disparity_func` to separate the implemented disparity functions;
- `utils.moravec_func` for the logic of the improvement 3.

```
1  import cv2
2  import numpy as np
3  import time
4  import argparse
5  import sys
6  from utils.disparity_func import disparity_function, precompute_ncc, precompute_zncc
7  from utils.draw_func import draw_planar_view
8  from utils.moravec_func import compute_moravec
```

Figure 2.1: Libraries used in the project

2.2 Inside the Main function

The main function is used to get arguments from the user to personalize program's parameters. They are put as arguments instead being taken from a configuration file for an easier implementation and customizable user experience. The eligible arguments are:

- **Metric**: (dis)similarity function used to compare pixels between stereo captures.
  Choices: **SAD**, **SSD**, **NCC**, **ZNCC**.
- **Aggregation**: method used to calculate the main disparity (d_main).
  Choices: **mean**, **median**, **mode**.
- **ROI Size**: size (pixels) of the central area of the reference frame.
- **Vertical Stripes**: number of vertical stripes desired for improvement 2.
- **Moravec Threshold**: value for filtering flat textures on the frame for improvement 3.
- **Video Left and Right**: path for left and right stereo videos (already rectified, as already said in the project specifics).

To avoid errors, due to internal integer divisions and calculation, there's a check to prevent overflows: (roi_size - window_size / 2) divided by vertical_stripes must have a discard of 0.

After taking all the external parameters, the main program is then run.

```
405     bot = RobotNavigation(metric=args.metric,
406                           aggregation=args.aggregation,
407                           roi_size=args.roi_size,
408                           window_size=args.window_size,
409                           vertical_stripes=args.vertical_stripes,
410                           moravec_threshold=args.moravec_threshold)
411
412     bot.run(args.video_left, args.video_right)
```

Figure 2.2: Passing of the parameters to the RobotNavigation and running with the videos.

2.3 Run method

The Run method consists on the endless cycle of RobotNavigation, composed by some tasks:

- **Capture** of the left and right frames from a `cv2.VideoCapture`.
- **Preprocess** left and right frames.
- Calculation of the frames parameters (e.g. x and y centers, Region of Interest (ROI) x and y starting coordinates etc.).
- Computation of the **disparity map**, along with the chosen **main disparity** and a **texture map** used to mask flat textures.
- Calculation of the **frontal distance** with the previously calculated **main disparity** value; focal lenght and baseline length are already given by the requirements.

- Check the calculated distance and **alarm** the user if the frontal distance is $<$ 0.8 meters.
- Calculating the **chessboard dimensions** if its corner are found by the opencv function `cv2.findChessboardCorners`.
- **Process the stripes** given a disparity map, returning a main disparity each stripe.
- Compute and draw the **angle** and the **planar view** given the previously calculated main disparities.
- **Draw** all the required frames.

All the tasks will be explained individually in the following sections.

2.4 Image preprocessing

Before proceeding with feature extraction, the frames must be preprocessed. This phase is crucial to standardize the data and optimize the input for the mathematical operations performed by the detection algorithm.

The preprocessing pipeline, implemented in the `preprocess_img` method, consists of two main steps: **grayscale conversion** and **edge sharpening**.

2.4.1 Grayscale Conversion

The first operation converts the image from the BGR color space to grayscale. As we already know, this reduction is necessary because algorithms such as Stereo Pixel Matching and Moravec Detector rely on pixel intensity variations rather than chromatic variations. Working with a single-channel (grayscale) image significantly reduces computational complexity without losing data.

In the code, this conversion is performed using the function

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

2.4.2 Image Sharpening

To improve the definition of edges and corners, a sharpening filter is applied to the grayscale image. This is achieved by convolving the image with a specific kernel designed to enhance high-frequency details (rapid changes in intensity).

The kernel used for this operation is defined as matrix $\mathbf{K}$:

$$\mathbf{K} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

In the code, this convolution is performed using the function

```
preprocessed = cv2.filter2D(gray, -1, self.kernel_sharpening)
```

Experimental results during the development phase highlighted a huge benefit by preprocessing images with sharpening. Applying this filter resulted in a much more linear distance estimation, significantly reducing fluctuations (noise) and improving overall stability. Furthermore, final tests confirmed that the Moravec detector also benefits from the enhanced edge contrast, performing slightly better.

```
49          # kernel for image preprocessing
50          self.kernel_sharpening = np.array([
51              [0, -1,  0],
52              [-1, 5, -1],
53              [0, -1,  0]
54          ], dtype=np.float32)
55
56      def preprocess_img(self, img):
57          gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
58          preprocessed = cv2.filter2D(gray, -1, self.kernel_sharpening)
59
60          return preprocessed
```

Figure 2.3: Declaration of the sharpening kernel and pre-processing function, executed on every frame pair

2.5 Disparity Computation

To maintain code modularity and ensure the main execution pipeline remains readable, the core mathematical operations for stereo matching and feature extraction were encapsulated in two external Python modules: `disparity_func.py` and `moravec_func.py`

2.5.1 Similarity and Dissimilarity Metrics

This module implements the mathematical functions required to compare pixel windows between the left and right images. It is designed to be agnostic to the specific stereo algorithm, allowing the user to switch between different metrics dynamically via `disparity_function`.

Instead of iterating through pixels using standard Python loops (which are computationally expensive), the functions utilize `cv2.boxFilter`. This OpenCV function uses hardware acceleration to calculate sums (and perform normalizations, if required) over sliding windows efficiently (whose size is specified in the arguments and passed through the call).

The module supports four distinct metrics, selectable via a string argument:

- **SAD (Sum of Absolute Differences)** and **SSD (Sum of Squared Differences)**: computationally cheap metrics based on intensity subtraction.
- **NCC (Normalized Cross-Correlation)** and **ZNCC (Zero-Mean NCC)**: robust statistical metrics that handle lighting variations but are computationally demanding.

An optimization was introduced for the normalized metrics (NCC/ZNCC). The denominator in these correlation formulas requires the calculation of the local sum of squares for the left image block. Since the left ROI remains static while we

search for matches across the disparity range, recalculating this value for every disparity shift would be a computational waste. The functions `precompute_ncc` and `precompute_zncc` calculate these invariant portions once before the disparity loop begins. This "one-shot" calculation reduces the computational load during the dense stereo matching phase.

2.5.2 Texture Detection and Filtering

Reliable stereo matching requires sufficient local texture; attempting to match flat, homogeneous regions (such as a uniform surface or a clear sky) results in noisy and incorrect disparity estimations. This module implements the **Moravec Interest Operator** to act as a confidence filter.

Algorithm Logic: The function `compute_moravec` analyzes the local contrast of a given ROI to determine if a pixel is suitable for matching.

1. **Directional Shifts**: The algorithm compares a central window of 3x3 pixels against shifted versions of itself in all 8 cardinal and ordinal directions (up, down, left, right, and diagonals).

2. **SSD Calculation**: For each direction, it calculates the **Sum of Squared Differences (SSD)** to quantify the intensity change.

3. **Minimization**: The algorithm identifies the minimum SSD value among all 8 directions. If the minimum SSD is high, it indicates that the pixel is distinct in every direction (a corner or strong texture). If the minimum SSD is low, the pixel belongs to a flat region or a simple edge (where sliding along the edge produces no change).

4. **Mask Generation**: Finally, the function applies a threshold to these minimum values, returning a boolean mask. This mask is used in the main pipeline to invalidate (ignore) pixels where the texture is insufficient for reliable depth estimation.

2.5.3 Main Function

The `compute_disparity` function represents the core of the stereo matching pipeline. It goes beyond simple disparity estimation by integrating temporal coherence and confidence filtering to ensure a robust calculation of the obstacle's distance. The process is divided into three logical stages:

Dynamic Search Window

Instead of scanning the entire disparity range for every frame, the algorithm exploits temporal coherence.

- **Dynamic Offset**: The search window is centered around the disparity value estimated in the previous frame (d_{prev}).

- **Optimization**: The algorithm searches a reduced local range (± 32 pixels) around the central value. This approach slightly reduces computational load and minimizes "jumps" caused by false positives in the background.

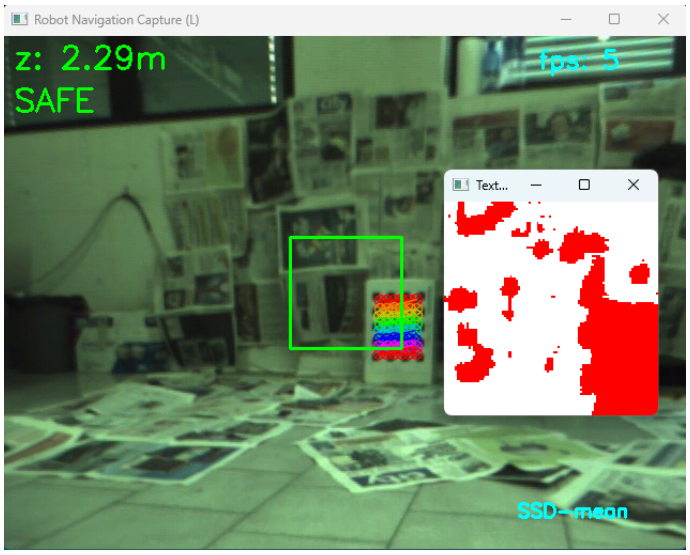

Figure 2.4: Texture Map (Moravec filter) applied to the ROI (green empty square in the bigger window). Red pixels represent valid areas (above the minimum texture threshold), which are used for disparity calculation. White pixels indicate areas where texture is insufficient, hence they are ignored.

```
64          half_disp_range = self.disp_range // 2
65          offset = max(0, int(self.d_prev - half_disp_range))
66          offset = min(offset, self.max_disp_value - self.disp_range)
67          self.dynamic_offset = offset
```

Figure 2.5: Dynamic Offset calculation procedure. Utilized to center the new range on the previous main disparity.

The central phase involves building a 3D Cost Volume of dimensions $(\mathbf{H}, \mathbf{W}, \mathbf{D})$, where $\mathbf{D}$ is the local disparity range.

- **Iterative Matching**: For each disparity step, the algorithm compares the Left ROI with the corresponding shifted Right ROI using the selected metric (SAD, SSD, NCC, or ZNCC). Pre-computed images are used here if normalized metrics are selected.
- **Selection**: Once the volume is populated, a strategy is applied along the depth axis. For every pixel (x,y), the algorithm selects the disparity index that minimizes (or maximizes) the cost.

Post-Processing and Aggregation

The raw disparity map contains noise and invalid border data. It undergoes a refinement process:

- **Border Cropping**: The edges of the disparity map (equal to half the window size) are removed, as convolution operations in these areas are invalid.
- **Texture Masking**: The Moravec Mask is applied. Pixels belonging to textureless regions are filtered out.

```
114         # we take the best indices for every y,x along depth (axis=2)
115         if self.maximize:
116             best_indices = np.argmax(cost_volume, axis=2)
117         else:
118             best_indices = np.argmin(cost_volume, axis=2)
119
120         # add the offset (best_indices are from 0 to 64)
121         disparity_map = best_indices.astype(np.float32) + offset
```

Figure 2.6: Selection of the indices (disparity values) corresponding to the best match along the depth axis. `np.argmax` is used for similarity metrics (maximum), while `np.argmin` is used for dissimilarity metrics (minimum).

- **Aggregation (d_{main})**: The dense map is condensed into a single scalar value, d_{main}, which represents the principal disparity of the detected obstacle. This value is calculated using statistical aggregation (mean, median, or mode) exclusively on the disparity values of the pixels validated by the texture mask. If the mask rejects all pixels (e.g., in a completely flat scene), the system falls back to the previous known disparity (d_{main}) to maintain stability.

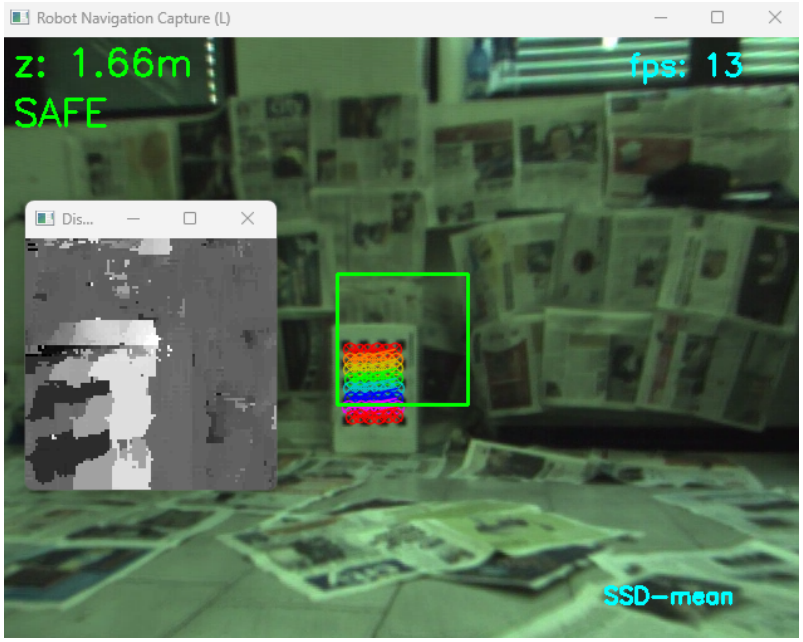

Figure 2.7: View of a Disparity Map example. In white pixels with a low (aggregated) disparity value, black viceversa.

2.6 Distance Calculation

Once the principal disparity (d_main) is determined, the pipeline proceeds to the final metric estimation and safety assessment:

- **Metric Conversion:** The principal disparity is passed to the project-defined `calculate_distance` method to obtain the depth Z in millimeters through

the formula:

$$z(mm) = \frac{b(mm) \cdot f(pixel)}{d_{main}(pixel)}$$

This value is immediately converted to meters to facilitate the threshold check.

- **Collision Warning Logic:** The system compares the estimated distance against a safety threshold of 0.8m.
  - If $Z < 0.8$, the system triggers an **ALARM** state, setting the visual feedback to red to alert the operator of an imminent collision.
  - Otherwise, the system remains in a **SAFE** state (green), indicating sufficient clearance.

```
294         d_main, disparity_map, texture_map = self.compute_disparity(gray_l, gray_r, roi_x, roi_y)
295
296         distance_mm = self.calculate_distance(d_main)
297         distance_m = distance_mm / 1000.0
298
299         color = (0, 255, 0)
300         alarm_text = "SAFE"
301         if distance_m < 0.8:
302             color = (0, 0, 255)
303             alarm_text = "ALARM! < 0.8m"
```

Figure 2.8: Portion of the code regarding the calculation of the distance and the safety alarm.

Figure 2.9: Visualization of the alarm when the robot gets below a 0.8 meters distance to the obstacle.

2.7 Chessboard Analysis

The main goal of this function is to perform a consistency check (or "sanity check") on the estimated distance. By detecting a calibration board with known dimensions, we can verify if the input z_distance is accurate.

The process follows three main steps:

1. **Pattern Detection**: First, the function converts the image to grayscale and uses `cv2.findChessboardCorners` to locate the internal corners of the chessboard pattern (defined as a 6×8 grid).

2. **Calculating Dimensions (Pixels)**: If the board is found, the code calculates its width and height in the image plane (in pixels). It uses `np.linalg.norm` to calculate the Euclidean distance between specific corners.

   - **Width (*wpix*)**: The distance between the top-left corner (index 0) and the top-right corner.
   - **Height (*hpix*)**: The distance between the top-left corner and the bottom-left corner.

   $$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

3. **Real World Dimension Estimation**: According to the formula given by the requirements, the physical dimensions in millimeters are calculated by applying the pinhole camera model principle (similar triangles), which scales the measured pixel dimensions based on the estimated **distance Z** and the **focal length f**.

```
Disparity range: [53, 117]    Disparity main: 88.72     Calculated dims.: w=129.4, h=181.7    (target: 125.0x178.0)
Disparity range: [56, 120]    Disparity main: 91.88     Calculated dims.: w=128.1, h=180.1    (target: 125.0x178.0)
Disparity range: [59, 123]    Disparity main: 94.26     Calculated dims.: w=127.7, h=179.4    (target: 125.0x178.0)
Disparity range: [62, 126]    Disparity main: 96.51     Calculated dims.: w=128.6, h=179.3    (target: 125.0x178.0)
Disparity range: [64, 128]    Disparity main: 97.49     Calculated dims.: w=130.4, h=181.8    (target: 125.0x178.0)
Disparity range: [64, 128]    Disparity main: 101.21    Calculated dims.: w=128.7, h=180.6    (target: 125.0x178.0)
Disparity range: [64, 128]    Disparity main: 103.37    Calculated dims.: w=128.3, h=178.4    (target: 125.0x178.0)
Disparity range: [64, 128]    Disparity main: 105.58    Calculated dims.: w=127.0, h=178.4    (target: 125.0x178.0)
Disparity range: [64, 128]    Disparity main: 106.93    Calculated dims.: w=128.4, h=179.3    (target: 125.0x178.0)
Disparity range: [64, 128]    Disparity main: 108.41    Calculated dims.: w=128.4, h=178.2    (target: 125.0x178.0)
Disparity range: [64, 128]    Disparity main: 108.69    Calculated dims.: w=128.8, h=179.0    (target: 125.0x178.0)
Disparity range: [64, 128]    Disparity main: 109.11    Calculated dims.: w=131.1, h=180.1    (target: 125.0x178.0)
Disparity range: [64, 128]    Disparity main: 108.62    Calculated dims.: w=131.1, h=180.1    (target: 125.0x178.0)
```

Figure 2.10: Verification Output. This output displays the estimated disparity and the comparison between the calculated target dimensions and its known real dimensions (target).

2.8 Stripes Processing

The `process_stripes` function implements a vertical segmentation method on the input disparity map. Its primary purpose is to simplify the disparity data by dividing the map into a user-defined number of vertical columns.

Inside the processing loop, the algorithm extracts the disparity data for each specific vertical section and computes the **arithmetic mean** (`np.mean`) of the pixel values. This operation results in a single, representative disparity value for the entire vertical segment, which is stored in the `d_stripes` list.

This segmentation creates a new, smoothed depth map that serves as the foundation for constructing a **planar view** (top-down perspective) of the robot's forward scene, where the average disparity of each column is converted into a specific metric distance (Z). Finally, by comparing the calculated distances of different columns (e.g., the difference between the left and right sectors), the system can estimate the surface angle and the **obstacle orientation** relative to the robot.

```
238     def process_stripes(self, disparity_map):
239         _, w = disparity_map.shape
240 
241         self.stripe_width = w // self.vertical_stripes
242         d_stripes = []
243         map_stripes = np.zeros_like(disparity_map)
244 
245         for i in range(self.vertical_stripes):
246             col_start = i * self.stripe_width
247             col_end = (i + 1) * self.stripe_width if i < self.vertical_stripes - 1 else w
248 
249             stripe_data = disparity_map[:, col_start:col_end]
250 
251             val = 0.0
252             if stripe_data.size > 0:
253                 # decided to use the classic mean to have smoother visualization of the stripes
254                 val = np.mean(stripe_data)
255 
256             map_stripes[:, col_start:col_end] = val
257             d_stripes.append(val)
258 
259         return d_stripes, map_stripes
```

Figure 2.11: Implementation of the `process_stripes` function. This function segments the disparity map, returning two key outputs: `d_stripes`, a list containing the mean disparity values used for planar reconstruction and obstacle orientation estimation, and `map_stripes`, the corresponding smoothed map utilized for simplified visualization.

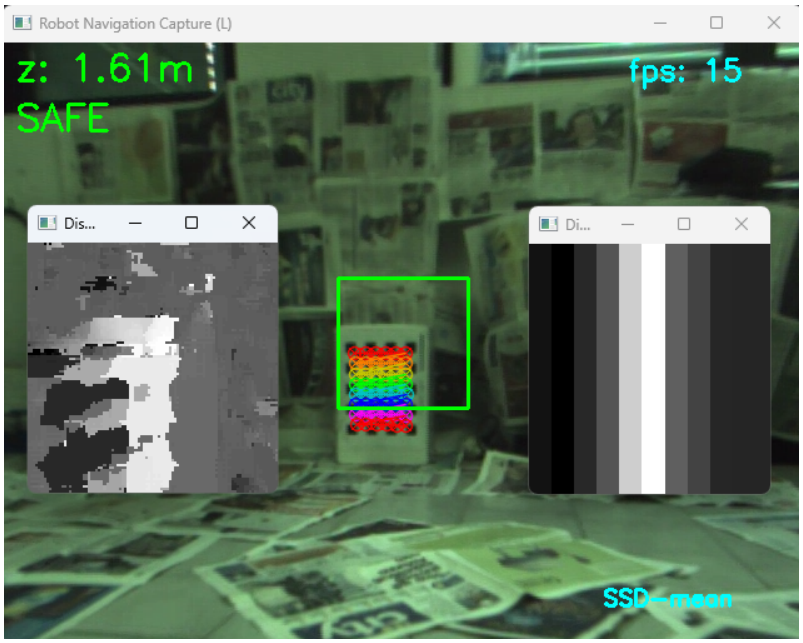

Figure 2.12: Visual Comparison of Disparity and Vertical Stripes Maps.

2.9 Planar View Computation

2.9.1 Stripes Positions

The `compute_planar_view` function is responsible for transforming the disparity data from the image plane (pixels) into real-world coordinates (millimeters) and recreating a planar view of the scene.

First, the function calculates the horizontal pixel coordinate (u) for the center of each stripe. After filtering out invalid disparity values, it applies the standard stereo vision triangulation formulas to reconstruct the scene:

- **Depth (z_{world})**: calculated using the focal length and baseline.
- **Lateral Position (x_{world}**: calculated by back-projecting the pixel coordinate u relative to the optical center.

The resulting (X, Z) pairs form a set of 2D points representing a "top-down" map of the scene, which is finally used to compute the obstacle's orientation angle.

2.9.2 Angle Calculation

The `calculate_obstacle_angle` function estimates the orientation of the observed surface relative to the camera's lateral axis.

The algorithm approximates the obstacle's surface as a linear segment connecting the two most extreme points available in the planar view: the first valid point (P_{start}) and the last valid point (P_{end}).

The orientation angle θ is then computed using the function (`arctan2`) based on the positional differences along the Z (depth) and X (lateral) axes:

$$\theta = \arctan 2(\Delta Z, \Delta X) = \arctan 2(Z_{end} - Z_{start}, X_{end} - X_{start}) \tag{2.1}$$

Finally, the result is converted from radians to degrees. A positive or negative angle indicates whether the obstacle is tilted towards or away from the robot.

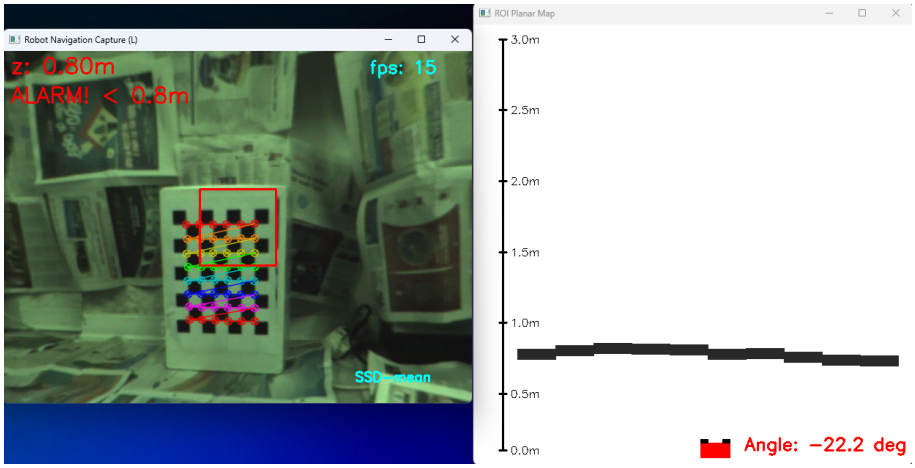

Figure 2.13: Planar Map Visualization for Metric Validation and Orientation Estimation. The 2D reconstruction displays the conformity of the processed points to the obstacle's actual geometry. This alignment visually confirms the accuracy of the calculated angle θ and the correctness of the triangulation formulas used for depth estimation.

2.10 Data Visualization and User Interface

The final stage of the processing pipeline involves rendering the computed data into multiple windows (seen before as samples) to provide real-time feedback on the robot's perception. The visualization logic is divided into three main categories:

- **Augmented Camera View (Frame L)**: The primary window displays the raw left frame from the stereo camera. It is augmented using `cv2.rectangle` to visualize the Region of Interest (ROI) and `cv2.putText` to overlay data, including the estimated distance (Z), the current frames per second (FPS), and any active collision alarms.
- **Intermediate Processing Maps**: In order to help in debugging and analysis, the system visualizes the internal states of the algorithm:
  1. **Disparity Map**: The raw normalized disparity map.
  2. **Stripes Map**: The vertically segmented map.
  3. **Texture Map**: A visualization of the texture analysis, where valid high-texture pixels are highlighted in red against a white background to verify the efficacy of the filtering mask.
- **Planar View Reconstruction**: Finally, the computed top-down map is displayed (ROI Planar Map). The calculated obstacle orientation angle is overlaid directly onto this map, providing an immediate metric reference for the obstacle's position and rotation relative to the robot.

Standard OpenCV window management (`cv2.namedWindow`, `cv2.resizeWindow`) is used to ensure the debug maps are scaled up for better visibility on the screen.

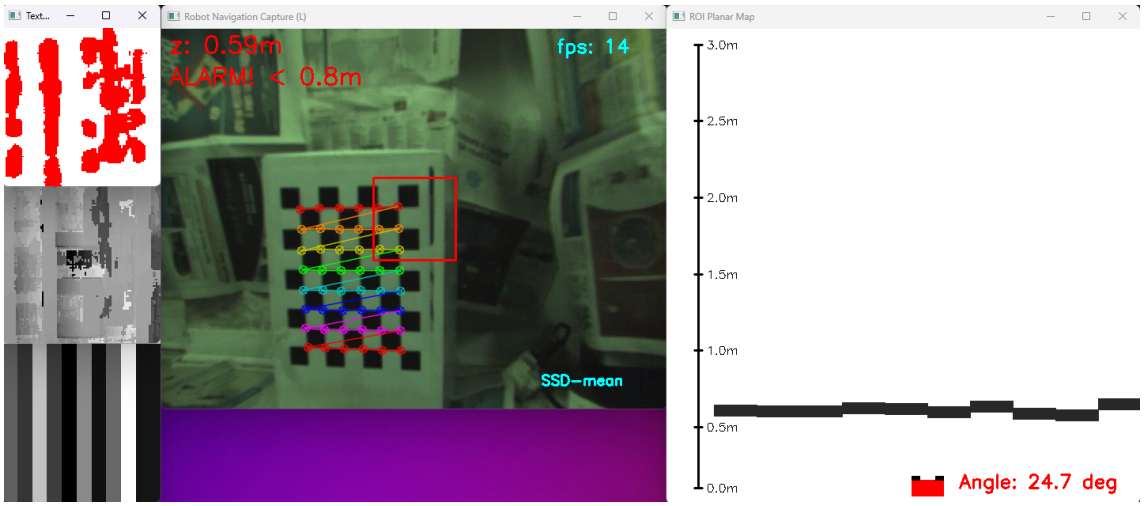

Figure 2.14: Complessive View of the program's output, chessboard's estimations (terminal output) excluded.

Chapter 3

Conclusion and Discussion of Results

Following an extensive testing phase conducted during the development of the program, it was possible to identify a parameter set that offers the ideal compromise between detection accuracy, robustness to noise, and computational efficiency.

The optimal configuration identified is as follows:

- **Matching Algorithm:** SSD (Sum of Squared Differences) with Mean aggregation
- **ROI Size:** 104 pixels
- **Window Size:** 9 (9×9 kernel)
- **Vertical Stripes:** 10
- **Moravec Threshold:** 11000

The following analysis details the motivations behind these choices.

3.1 Choice of Stereo Matching Algorithm

Although metrics such as NCC (Normalized Cross Correlation) and ZNCC (Zero-mean NCC) are renowned for their robustness to illumination variations, they proved to be computationally expensive without providing significant benefits in this specific context. Since the system operates in real-time stereo mode, capturing the scene simultaneously from both lenses, the lighting conditions are virtually identical across the views. Consequently, the **SSD** is been chosen, because it offers a slightly better balance between speed and accuracy, heavily penalizing large correspondence errors, which is consistent with the goal of having a robust system, especially in the proximity of an obstacle.

3.2 ROI and Window Sizing

The **ROI size of 104** was calibrated to monitor a central yet sufficiently extensive area. This dimensioning is crucial during the approach phase to an obstacle: a too

small ROI risks being saturated by flat (textureless) zones when the object is very close. The chosen size ensures the presence of sufficient features to accurately detect distances below the critical alarm threshold (< 0.8 meters).

In parallel, the **Window Size of** 9×9 represents the ideal equilibrium point:

- A smaller window would be too sensitive to noise, generating false matches.
- A larger window would excessively reduce the specificity of the matching, oversmoothing the information.

3.3 Geometric Constraints and Planar View

The value for the subdivision of the scene into **Vertical Stripes** was set to **10**. Compared to a coarser subdivision (4-5 stripes), this value entails a slight increase in computational load but offers superior resolution in the *Planar View*, allowing for a much better morphological understanding of the obstacle.

Furthermore, the combination of geometric parameters strictly respects the integer condition:

```
(roi_size - (window_size // 2)) % vertical_stripes == 0
```

This constraint ensures that the subdivision of calculation areas occurs without remainders, avoiding artifacts or rounding errors at the stripe boundaries.

3.4 Feature Extraction Robustness (Moravec)

The Moravec operator threshold was experimentally fixed at **11000**. Tests highlighted that higher thresholds tended to select almost exclusively pixels belonging to the calibration chessboard (characterized by sharp edges). A threshold of 11000 allows the mask to include pixels belonging to the background or other objects as well. This makes the system significantly more robust, avoiding overfitting on high-contrast features and functioning correctly even when the robot looks outside the chessboard area.

3.5 Performance Evaluation

The system demonstrates good overall accuracy, with slight fluctuations in distance measurement that fall within expected tolerances. The behavior is optimal at short distances, while a slight degradation of precision is observed at long ranges.

Due to the intensive calculations, the operational frame rate settles between **5 and 8 fps** (compared to the original 15 fps of the video). However, an interesting behavior was observed: in close proximity to an obstacle, the system recovers fluidity, reaching maximum fps and ensuring maximum reactivity precisely during the moments of a potential collision.

3.5.1 Impact of Image Pre-processing (Sharpening)

Experimental results highlighted the critical role of image sharpening in the processing pipeline. Tests conducted without this pre-processing step gave significantly less accurate depth estimations.

The application of a sharpening kernel proved to be essential for stabilizing the measurements: by enhancing the edge details within the image, the (dis)similarity algorithms can identify correspondence points with greater certainty. This results in a reduction of measurement variance and a more consistent distance output over time.